
PROMINENCE User Manual

Jul 14, 2022

Contents:

1	Introduction	3
1.1	Workflows, jobs and tasks	3
1.2	Job lifecycle	5
2	Containers	7
2.1	Why containers?	7
2.2	Accessing images	7
2.3	Tips for creating containers	8
2.4	MPI jobs	8
2.5	Registry authentication	9
3	Command line interface	11
3.1	Installation	11
3.2	Setup	12
3.3	Help	12
3.4	Prerequisites	12
3.5	Getting an access token	13
3.6	Your first job	13
3.7	Jobs and workflows	14
4	Jobs	17
4.1	Running jobs	17
4.2	Standard output and error	19
4.3	Checking job status	20
4.4	Deleting a job	21
4.5	Labels	21
4.6	Generating JSON	22
4.7	Policies	23
4.8	Environment variables	24
4.9	Inputs	25
5	Workflows	27
5.1	Types of workflows	27
5.2	Failures and retries	31
6	Data	33
6.1	B2DROP	33

6.2	OneData	34
6.3	WebDAV	34
7	Using the API	35
7.1	cURL	35
7.2	Python	39
8	Examples	41
8.1	Jobs	41
8.2	Workflows	41
8.3	Jupyter notebooks	45

PROMINENCE is a platform which allows users to exploit idle cloud resources for running containerised workloads. From a user's perspective PROMINENCE appears like a standard batch system but jobs can be run from anywhere in the world on clouds anywhere in the world.

Computing resources are provided by [EGI Federated Cloud](#) sites.

Features include:

Flexible submission

- Submit jobs using a simple batch system style command line interface which can be installed anywhere
- Interact programmatically from any language using a RESTful API
- Submit jobs from any Jupyter notebook

Reliability and reproducibility

- All jobs are run in containers to ensure they will reliably run anywhere and are reproducible

Multi-cloud native

- Go beyond a single cloud and leverage the resources and features available across many clouds

Invisible infrastructure

- All infrastructure provisioning is handled completely automatically and is totally transparent to the user

Workflows

- Construct complex workflows by specifying the dependencies between different jobs
- Automatically perform parameter sweeps

MPI ready

- Run multi-node Open MPI, Intel MPI or MPICH jobs in addition to single-node HTC jobs

1.1 Workflows, jobs and tasks

A **job** in PROMINENCE consists of the following:

- Name
- Labels

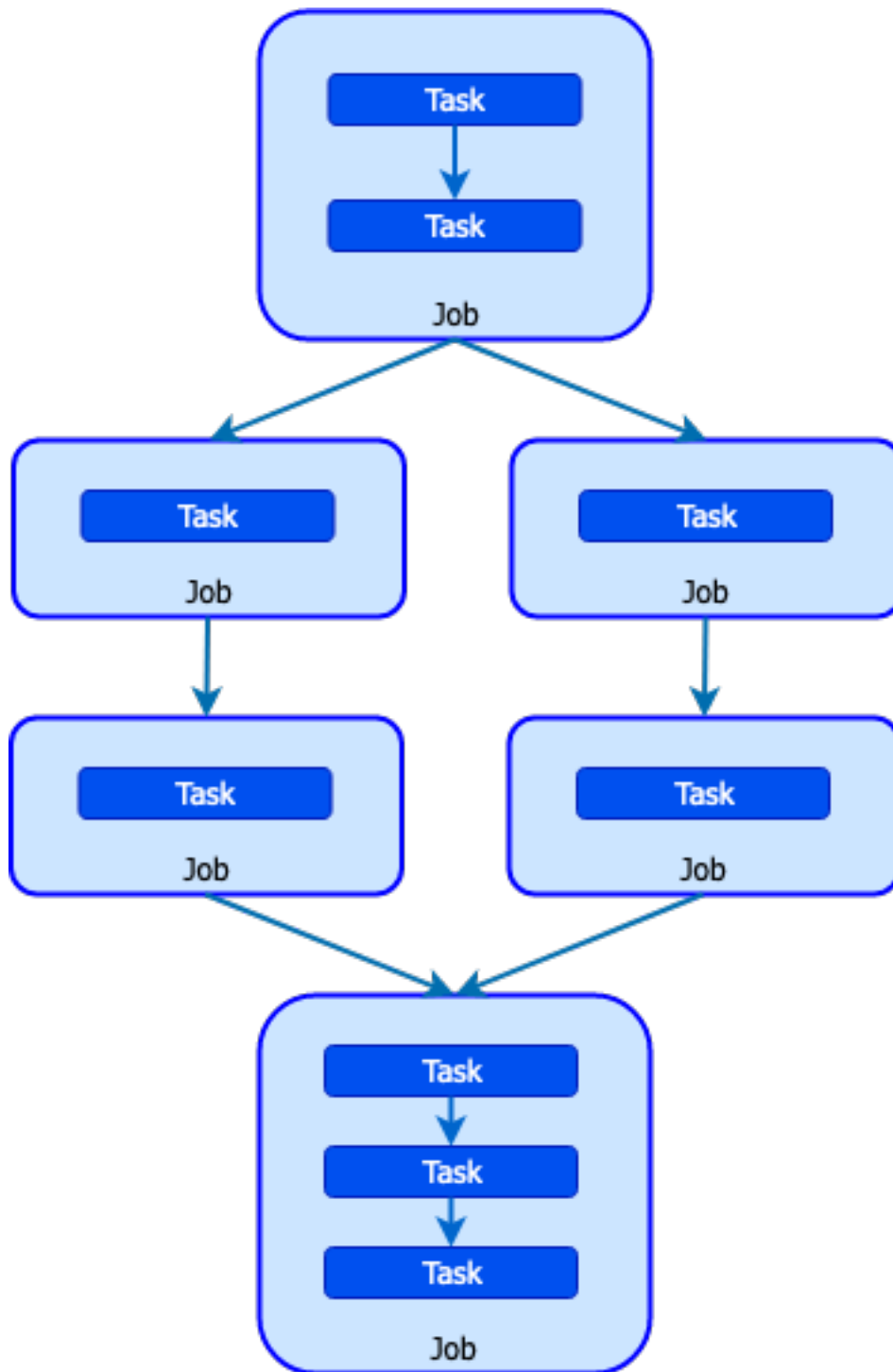
- Input files
- Storage
- Required resources (e.g. CPU cores, memory, disk)
- One or more task definitions (described below)
- Policies (e.g. how many times should failing tasks should be retried)

Tasks execute sequentially within a job, and consist of the following:

- Container image
- Container runtime
- Command to run and optionally any arguments
- Environment variables
- Current working directory

A **workflow** consists of one or more jobs and optionally any dependencies between them. Jobs within a workflow can be executed sequentially, in parallel or combinations of both.

An example workflow, including how it is made up of jobs and tasks, is shown below:



1.2 Job lifecycle

List of possible job states:

- **Pending:** the job is not yet running.
- **Running:** the job is running.

- **Completed:** the job has completed, however note that the user application's exit status may or may not be 0.
- **Deleted:** the job has been deleted by the user.
- **Killed:** the job has been forcefully terminated, for example it had been running for too long.
- **Failed:** the job failed, for example the container image could not be pulled.

In PROMINENCE all jobs are run in unprivileged containers using user-specified images. It is possible to use either the Singularity or udocker container runtimes, but of course standard Docker images can be used.

2.1 Why containers?

Every job in PROMINENCE needs to have a container image specified. Running jobs in containers ensure that the application will be able to run successfully without depending on any software on the host. This is important because PROMINENCE makes use of a variety of different cloud resources, managed by different people from different organizations, potentially around the world.

2.2 Accessing images

The image can be specified in the following ways:

- `<user>/<repo>:<tag>` (Docker Hub)
- `<hostname>/<project-id>/<image>:<tag>` (Google Container Registry)
- `shub://<user>/<repo>:<tag>` (Singularity Hub)
- URL of a tarball created by `docker save`
- URL of a Singularity image
- path of a tarball created by `docker save` when remote storage is attached to jobs
- path of a Singularity image when remote storage is attached to jobs

Container registries other than Docker Hub may also work. It is possible to provide credentials to use for authenticating with a registry.

Under some conditions a container runtime will be selected automatically. This will only happen if there is only one runtime which will work for the specified image. For other cases, e.g. a Docker Hub image, Singularity is used as the default but optionally udocker can be forced by the user.

Images which will result in Singularity being selected:

- Singularity Hub image (begins with `shub://`)
- URL for a Singularity image (filename ends in `.sif` or `.simg`)

Images which will result in udocker being selected:

- URL for a Docker tarball (filename ends in `.tar`)

Generally it is recommended to use Singularity. If it is essential to be able to write into the container's filesystem (e.g. `/opt` or `/var`) then udocker should be used.

2.3 Tips for creating containers

Some important tips for creating containers to be used with PROMINENCE:

- Do not put any software or required files in `/root`, since containers are run as an unprivileged user.
- Do not put any software or required files in `/home` or `/tmp`, as these directories in the container image will be replaced when the container is executed.
- Do not specify `USER` in your Dockerfile when creating the container image.
- The environment variable `HOME` will be set to a scratch directory (`/home/user`) accessible inside the container when the container is executed. For the case of multi-node MPI jobs this scratch directory is accessible across all nodes running the job.
- The environment variables `TMP` and `TEMP` are set to `/tmp`. This directory is always local to the host, including for multi-node MPI jobs.
- Do not expect to be able to write inside the container's filesystem. Write any files into the default current working directory, or into the directories specified by the environment variables `HOME`, `TMP` and `TEMP`.
- The application should be able to be run from within any directory and access any required input or output files using relative paths.

2.4 MPI jobs

Some are some additional requirements on the container images for MPI jobs:

- `mpirun` should be available inside the container and in the `PATH`
- The `ssh` command should be installed inside the container

There is no reason to set an entrypoint as it will not be used. A command (and any required arguments) must be specified.

A simple minimal starting point for a Dockerfile for a CentOS 7 container image for OpenMPI is:

```
FROM centos:7
RUN yum -y install openssh-clients openmpi openmpi-devel

ENV PATH /usr/lib64/openmpi/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/lib64/openmpi/lib:${LD_LIBRARY_PATH}
```

and for MPICH:

```
FROM centos:7
RUN yum -y install openssh-clients mpich mpich-devel

ENV PATH /usr/lib64/mpich/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/lib64/mpich/lib:${LD_LIBRARY_PATH}
```

To create a container using IntelMPI an Intel compiler licence is required to build the application. This application can then be copied into a container image with the Intel Parallel Studio Runtime installed. For example, see [here](#) for information on installing the free Intel runtime in a CentOS environment.

2.5 Registry authentication

If a registry requires authentication then `imagePullCredential` must be specified in the task, which defines a username and token. For example:

```
{
  "resources": {
    "memory": 1,
    "cpus": 1,
    "nodes": 1,
    "disk": 10
  },
  "name": "gitlab-image",
  "tasks": [
    {
      "image": "registry.gitlab.com/mynamespace/myproject/image:rc1",
      "runtime": "singularity",
      "imagePullCredential": {
        "username": "username",
        "token": "VzIxo3sZ2yC6V5YeSBxR"
      }
    }
  ]
}
```

The same method can be used for other private registries and Docker Hub. It is preferable that a token with read-only privileges is used rather than a password if at all possible.

Command line interface

3.1 Installation

The PROMINENCE CLI can be installed from PyPI. It is possible for normal users to install the PROMINENCE CLI without having to request any assistance from their system administrators.

3.1.1 With sudo or root access

The PROMINENCE CLI can be installed on a host by typing the following:

```
$ sudo pip install prominence-cli
```

3.1.2 As a normal user without using a virtual environment

The PROMINENCE CLI can be installed in a user's home directory by running:

```
$ pip install --user prominence-cli
```

The directory `.local/bin` will need to be added to the `PATH`.

3.1.3 As a normal user using a virtual environment

The PROMINENCE CLI can be installed in a new virtual environment, e.g.

```
$ mkdir ~/.virtualenvs
$ python3 -m venv ~/.virtualenvs/prominence
$ source ~/.virtualenvs/prominence/bin/activate
$ pip install prominence-cli
```

If `pip` is not available for some reason it can be installed in a user's home directory by typing the following before running the above:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3 get-pip.py --user
$ pip install --user virtualenv
```

Checking what version is installed

The version of the PROMINENCE CLI installed can be checked by running:

```
$ prominence --version
```

3.2 Setup

Create a configuration directory for PROMINENCE:

```
$ mkdir ~/.prominence
```

3.3 Help

The main help page gives a list of all the available commands:

```
$ prominence --help
usage: prominence [-h] [--version]
                  {run,create,list,describe,delete,stdout,stderr} ...

Prominence - run jobs in containers across clouds

positional arguments:
  {run,create,list,describe,delete,stdout,stderr}
                                sub-command help
  create                    Create a job
  run                       Create a job or workflow from JSON or YAML in a file or URL
  list                      List jobs or workflows
  describe                  Describe a job or workflow
  delete                    Delete a job or workflow
  stdout                    Get standard output from a running or completed job
  stderr                    Get standard error from a running or completed job

optional arguments:
  -h, --help                show this help message and exit
  --version                  show the version number and exit
```

3.4 Prerequisites

In order to use the EGI PROMINENCE service you must be a member of one of the following VOs and have the **vm_operator** role:

- vo.access.egi.eu

- fusion

Support for additional VOs can be added upon request.

3.5 Getting an access token

In order to interact with the PROMINENCE service via the CLI an access token is required. Go to the <https://eosc.prominence.cloud> and click on **Login** to log in with your Check-in credentials to obtain an access token.

The FedCloud Check-in client also provides the exact command to run to generate an access token. The PROMINENCE CLI requires the output of this command to be stored in the file `~/.prominence/token`. The command to run will be of the form:

```
$ curl -X POST -u '<client id>': '<client secret>' -d 'client_id=<client id>&client_secret=<client secret>&grant_type=refresh_token&refresh_token=<refresh token>&scope=openid%20email%20profile' 'https://aai.egi.eu/oidc/token' > ~/.prominence/token
```

Note: Since PROMINENCE uses a REST API every request needs to be authenticated and hence requires an access token. The access token is not stored in any way by the server.

3.6 Your first job

Run the following command in order to submit a simple test job by typing `prominence create eoscprominence/testpi`, i.e.

```
$ prominence create eoscprominence/testpi
Job created with id 7375
```

Here `eoscprominence/testpi` is the name of the container image on Docker Hub. This command will submit a job which runs a container from the `eoscprominence/testpi` image using the default entrypoint specified in the image. In this case it is a simple Python script which calculates pi in three different ways.

To check the status of the job, run `prominence list` to list all currently active jobs:

```
$ prominence list
```

ID	NAME	CREATED	STATUS	ELAPSED	IMAGE	CMD
7375		2019-10-14 12:33:11	pending		eoscprominence/testpi	

Eventually the status will change to `running`, `completed` and then disappear. The `list` command can be given the argument `--completed` to show completed jobs. For example, to see the most recently completed job:

```
$ prominence list --completed
```

ID	NAME	CREATED	STATUS	ELAPSED	IMAGE	CMD
7375		2019-10-14 12:33:11	completed	0+00:00:15	eoscprominence/testpi	

Once the test job has finished running, `prominence stdout` can be used to view the standard output, e.g.

```
$ prominence stdout 7375
```

Plouff	Bellard	Chudnovsky
--------	---------	------------

(continues on next page)

(continued from previous page)

[illegible]

The next sections of the documentation describe in more detail how to run more complex jobs and workflows.

3.7 Jobs and workflows

By default all PROMINENCE CLI commands refer to jobs. However, a number of commands include the ability to specify a resource, which is either a `job` or a `workflow`.

Listing workflows:

```
prominence list workflows
```

Describing a specific workflow:

```
prominence describe workflow <id>
```

Deleting a workflow:

```
prominence delete workflow <id>
```

The standard output and error from a job which is part of a workflow can be viewed by specifying both the workflow id and the name of the job, i.e.

```
prominence stdout <id> <job name>
```

To list all the individual jobs from workflow <id>:

```
prominence list jobs <id>
```


4.1 Running jobs

4.1.1 Single node jobs

In order to run an instance of a container, running the command defined in the image's entrypoint, all you need to do is to specify the Docker Hub image name:

```
$ prominence create eoscprominence/testpi
Job created with id 3101
```

When a job has been successfully submitted an (integer) ID will be returned. Alternatively, a command (and arguments) can be specified. For example:

```
$ prominence create centos:7 "/bin/sleep 100"
```

The command of course should exist within the container. If arguments need to be specified you should put the command and any arguments inside a single set of double quotes, as in the example above.

To run multiple commands inside the same container, use `/bin/bash -c` with the commands enclosed in quotes and separated by semicolons, for example:

```
$ prominence create centos:7 "/bin/bash -c \"date; sleep 10; date\""
```

This is of course assuming `/bin/bash` exists inside the container image.

4.1.2 MPI jobs

To run an MPI job, you need to specify either `--openmpi` for Open MPI, `--intelmpi` for Intel MPI and `--mpich` for MPICH. For multi-node jobs the number of nodes required should also be specified. For example:

```
$ prominence create --openmpi --nodes 4 alahiff/openmpi-hello-world:latest /mpi_hello_
↪world
```

The number of processes to run per node is assumed to be the same as the number of cores available per node. If the number of cores available per node is more than the requested number of cores all cores will be used. This behaviour can be changed by using `--procs-per-node` to define the number of processes per node to use.

Note: It is always preferable to run MPI jobs within a single node if possible, especially for low numbers of CPU cores.

For MPI jobs a command to run (and optionally any arguments) must be specified. If an entrypoint is defined in the container image it will be ignored.

4.1.3 Hybrid MPI-OpenMP jobs

In this situation the number of MPI processes to run per node must be specified using `--procs-per-node` and the environment variable `OMP_NUM_THREADS` should be set to the required number of OpenMP threads per MPI process.

In the following example we have 2 nodes with 4 CPUs each, and we run 2 MPI processes on each node, where each MPI process runs 2 OpenMP threads:

```
$ prominence create --cpus 4 \  
                  --memory 4 \  
                  --nodes 2 \  
                  --procs-per-node 2 \  
                  --openmpi \  
                  --env OMP_NUM_THREADS=2 \  
                  --artifact https://github.com/lammps/lammps/archive/stable_  
↪12Dec2018.tar.gz \  
                  --workdir lammps-stable_12Dec2018/bench \  
                  alahiff/lammps-openmpi-omp "lmp_mpi -sf omp -in in.lj"
```

4.1.4 Resources

By default a job will be run with 1 CPU and 1 GB memory but this can easily be changed. The following resources can be specified:

- CPU cores
- Memory (in GB)
- Disk (in GB)
- Maximum runtime (in mins)

CPU cores and memory can be specified using the `--cpus` and `--memory` options. A disk size can also be specified using `--disk`.

Here is an example running an MPI job on 4 nodes where each node has 2 CPUs and 8 GB memory, there is a shared 20 GB disk accessible by all 4 nodes, and the maximum runtime is 1000 minutes:

```
$ prominence create --openmpi \  
                  --nodes 4 \  
                  --cpus 2 \  
                  --memory 8 \  
                  --disk 20 \  
                  --runtime 1000 \  
                  alahiff/geant4mpi:1.3a3
```

By default a 10 GB disk is available to jobs, which is located on separate block storage. For MPI jobs the disk is available across all nodes running the job. The default maximum runtime is 720 minutes.

4.1.5 Working directory

By default the current working directory is scratch space made available inside the container. The path to this directory is also specified by the environment variables HOME, TEMP and TMP.

To specify a different working directory use `--workdir`. For example, the following will run `pwd` inside the `/tmp` directory.

```
$ prominence create --workdir /tmp centos:7 pwd
```

Note: Remember that you should not try to write inside the container’s filesystem as this may be prevented by the container runtime or result in permission problems.

4.1.6 Environment variables

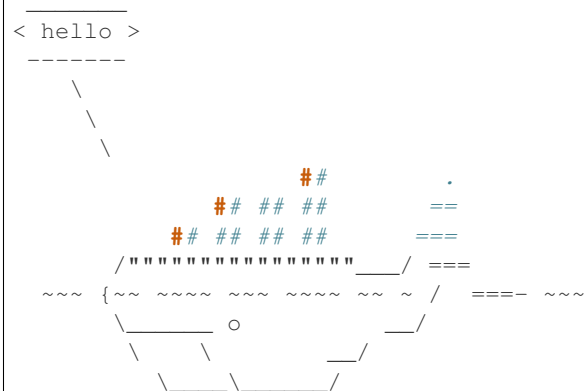
It is a common technique to use environment variables to pass information, such as configuration options, into a container. The option `--env` can be used to specify an environment variable in the form of a key-value pair separated by `=`. This option can be specified multiple times to set multiple environment variables. For example:

```
$ prominence create --env LOWER=4.5 --env UPPER=6.7 test/container
```

4.2 Standard output and error

The standard output and error from a job can be seen using the `stdout` and `stderr` commands. For example, to get the standard output for the job with id 299:

```
$ prominence stdout 299
```



Note: The standard output and error can be seen while jobs are running as well as once they have completed, allowing users to check the status of long-running jobs.

4.3 Checking job status

4.3.1 Listing jobs

The list command by default lists any active jobs, i.e. jobs which are pending or running:

```
$ prominence list
ID      STATUS  IMAGE                                CMD      ARGS
3101    pending  alahiff/testpi
3103    pending  alahiff/cherab-jet:latest          python  batch_make_sensitivity_matrix.py
↪0 59
3104    pending  ikester/blender:latest             blender -b classroom/classroom.blend -o
↪frame_### -f
```

It's also possible to request a list of jobs using a constraint on the labels associated with each job. For example, if you submitted a group of jobs with a label `name=run5`, the following would list all such jobs:

```
$ prominence list --all --constraint name=run5
```

Here the `--all` option means that both active (i.e. pending or running) and completed jobs will be listed.

4.3.2 Describing a job

To get more information about an individual job, use the describe command, for example:

```
$ prominence describe 345
{
  "id": ,
  "status": "pending",
  "resources": {
    "nodes": 1,
    "disk": 10,
    "cpus": 1,
    "memory": 1
  },
  "tasks": [
    {
      "image": "eoscprominence/testpi",
      "runtime": "singularity"
    }
  ],
  "events": {
    "createTime": "2019-10-04 18:07:40"
  }
}
```

To show information about completed jobs, both the list and describe commands accept a `--completed` option. For example, to list the last 2 completed jobs:

```
$ prominence list --completed --last 2
ID      STATUS  IMAGE                                CMD      ARGS
2980    completed  alahiff/tensorflow:1.11.0          python  models-1.11/official/
↪mnist/mnist.py --export_dir mnist_saved_model
2982    completed  alahiff/tensorflow:1.11.0          python  models-1.11/official/
↪mnist/mnist.py --export_dir mnist_saved_model
```


Note that jobs which are completed or have been removed for some reason may be visible briefly without using the `--completed` option.

4.3.3 Completed jobs

The JSON descriptions of completed jobs contain additional information. This may include:

- **status**: current job status.
- **statusReason**: for jobs in a terminal state other than the completed state this may give a reason for the current status.
- **createTime**: date & time when the job was created by the user.
- **startTime**: date & time when the job started running.
- **endTime**: date & time when the job ended.
- **site**: the site where the job was executed.
- **maxMemoryUsageKB**: the maximum total memory usage of the job, summed over all processes (note this is not available for jobs running on remote HTC or HPC resources)

The following information is also provided for each task:

- **retries**: the number of retries attempted.
- **exitCode**: the exit code returned by the user's job. This would usually be 0 for success.
- **imagePullTime**: time taken to pull the container image. If a cached image from a previous task was used this will be -1.
- **wallTimeUsage**: wall time used by the task.
- **cpuTimeUsage**: CPU time usage by the task. For a task using multiple CPUs this will be larger than the wall time.
- **maxResidentSetSizeKB**: maximum resident size (in KB) of the largest process

4.4 Deleting a job

Jobs cannot be modified after they are created but they can be deleted. The delete command allows you to kill a single job:

```
$ prominence delete 164
Success
```

4.5 Labels

Arbitrary metadata in the form of key-value pairs can be associated with jobs.

4.5.1 Defining labels

Labels in the form of key-value pairs (separated by "=") can be set using the `--label` option when creating a job. This option can be used multiple times to set multiple labels. For example:

```
$ prominence create --label experiment=MAST-U --label shot=12 test/container
```

Each key and value must be a string of less than 64 characters. Keys can only contain alphanumeric characters ([a-z0-9A-Z]) while values can also contain dashes (-), underscores (_), dots (.) and forward slashes (/).

4.5.2 Finding jobs with specific labels

It is possible to specify a constraint when using the list command. For example, to list all active jobs which have a label `experiment` set to `MAST-U`:

```
$ prominence list --constraint experiment=MAST-U
```

To list all jobs, i.e. both active and completed jobs, with a specific label, add the `-all` option, e.g.

```
$ prominence list --constraint experiment=MAST-U --all
```

4.6 Generating JSON

When `prominence create` is run with the `--dry-run` option, the job will not be submitted but the JSON description of the job will be printed to standard output. For example:

```
$ prominence create --dry-run --name test1 --cpus 4 --memory 8 --disk 20 busybox
{
  "resources": {
    "memory": 8,
    "cpus": 4,
    "nodes": 1,
    "disk": 20
  },
  "name": "test1",
  "tasks": [
    {
      "image": "busybox",
      "runtime": "singularity"
    }
  ]
}
```

If the JSON output is saved in a file it be submitted to PROMINENCE using the `run` command, e.g.:

```
$ prominence run <filename.json>
```

The job description can also be a URL rather than a file, e.g.

```
$ prominence run <https://.../filename.json>
```

4.6.1 Multiple tasks in a single job

By default a job will run a single command inside a single container. However, it is possible to instead run multiple sequential tasks within a single job. Each task will have access to the same temporary storage, so transient files generated by one task are accessible by other tasks.

To run multiple tasks it is necessary to construct a JSON description of the job. For example, in this job there are two sequential tasks:

```
{
  "resources": {
    "memory": 1,
    "cpus": 1,
    "nodes": 1,
    "disk": 10
  },
  "name": "multiple-tasks",
  "tasks": [
    {
      "image": "centos:7",
      "runtime": "singularity",
      "cmd": "cat /etc/redhat-release"
    },
    {
      "image": "centos:8",
      "runtime": "singularity",
      "cmd": "cat /etc/redhat-release"
    }
  ]
}
```

Use of JSON job descriptions is also necessary to run workflows, which we will come to next.

4.7 Policies

The `policies` section of a job's JSON description enables users to have more control of how jobs are managed and influence where they will be executed. The available options are:

- **maximumRetries:** maximum number of times a failing job will be retried.
- **maximumTimeInQueue:** maximum time in minutes the job will stay in the `pending` state. The default is 0, which means that the job will stay in the `pending` state until it starts running or there is a failure. The value `-1` means that the job will stay in the `pending` state until it starts running. When set to a non-zero value, the job will be automatically set to the `failed` state if it has not started running within the time limit.
- **leaveInQueue:** when set to `true` (default is `false`) completed, failed and deleted jobs will remain in the queue until the user specifies that they can be removed.
- **placement:** allows users to specify requirements and preferences to influence where jobs will run.

For example:

```
{
  "tasks": [
    {
      "image": "centos:7",
      "runtime": "singularity",
      "cmd": "date"
    }
  ],
  "name": "test",
  "resources": {
    "nodes": 1,

```

(continues on next page)

(continued from previous page)

```
"disk": 10,
"cpus": 1,
"memory": 1
},
"policies": {
  "maximumRetries": 4,
  "maximumTimeInQueue": 600,
  "leaveInQueue": true
}
}
```

4.7.1 Placement policies

Job placement policies enable users to influence where jobs will be executed. Sites refer to specific clouds and regions refer to groups of clouds. Placement policies consist of `requirements` (which must be satisfied) and `preferences` (used for ranking).

To force a job to run at a particular site (OpenStack-Alpha in this case):

```
"policies": {
  "placement": {
    "requirements": {
      "sites": [
        "OpenStack-Alpha"
      ]
    }
  }
}
```

To force a job to run at any site in a particular region:

```
"policies": {
  "placement": {
    "requirements": {
      "regions": [
        "Alpha"
      ]
    }
  }
}
```

4.8 Environment variables

Some environment variables are set automatically and are available for jobs to use.

- **PROMINENCE_CPUS**: the number of CPUs available, which could be larger than what was requested
- **PROMINENCE_MEMORY**: the amount of memory in GB available, which could be larger than what was requested
- **PROMINENCE_CONTAINER_RUNTIME**: the container runtime in use, either `singularity` or `udocker`
- **PROMINENCE_JOB_ID**: the id of the job

- **PROMINENCE_WORKFLOW_ID**: the id of the associated workflow, if applicable
- **PROMINENCE_URL**: URL of the PROMINENCE REST API
- **PROMINENCE_TOKEN**: token which can be used to authenticate against the PROMINENCE REST API (a unique token is generated per job, and is valid for the lifetime of the job)

4.9 Inputs

Most jobs of course require input files of some kind.

4.9.1 Input files

Small input files can be uploaded from the host running the CLI and made available to jobs using the *-input* option. For example:

```
prominence create --input README centos:7 "cat README"
```

The files will be written in the job's default current directory, also referred to by the HOME, TMP or TEMP environment variables. *-input* can be specified multiple times for multiple input files.

Note that large data files should not be provided to jobs this way, and there is a total size limit of 1 MB per file.

4.9.2 Artifacts

Input files, including much large files if necessary, can be obtained from standard URLs using the *-artifact* option. This enables input files to be downloaded from web servers or from object storage using presigned URLs.

```
prominence create --input https://raw.githubusercontent.com/lammps/lammps/develop/  
→examples/ASPHERE/star/in.star centos:7 "cat in.star"
```


5.1 Types of workflows

5.1.1 Groups of jobs

In some situations it can be useful to be able to manage a group of jobs as a single entity. In order to submit a group of independent jobs as a workflow the first step is to write a JSON description of the workflow. This is just a list of the definitions of the individual jobs, which can be created easily using `prominence create --dry-run`. The basic structure is:

```
{
  "name": "test-workflow-1",
  "jobs": [
    {...},
    {...}
  ]
}
```

5.1.2 Directed acyclic graphs

In order to submit a workflow the first step is to write a JSON description of the workflow. This is just a list of the definitions of the individual jobs, each of which can be created easily using `prominence create --dry-run`, along with the dependencies between them. Each dependency defines a parent and its children. The basic structure is:

```
{
  "name": "test-workflow-1",
  "jobs": [
    {...},
    {...}
  ],
  "dependencies": {
```

(continues on next page)

(continued from previous page)

```

    "parent_job": ["child_job_1", ...],
    ...
  }
}

```

Each of the individual jobs must have defined names as these are used in order to define the dependencies. Dependencies between jobs are *abstract dependencies*, i.e. defined in terms of names of jobs. This is different to CWL, for example, where the dependencies arise due to input and output data or files required by each job.

It is important to note that the resources requirements for the individual jobs can be (and should be!) specified. This will mean that each step in a workflow will only use the resources it requires. Jobs within a single workflow can of course request very different resources, which makes it possible for workflows to have both HTC and HPC steps.

5.1.3 Job factories

Job factories allow many similar jobs to be created from a single template. The following types of job factories are available:

- **parameter sweep**: a set of jobs is created by sweeping one or more parameters through a range of values
- **zip**: a set of jobs is created from multiple lists, where the i-th job contains the i-th element from each list
- **repeat**: runs the same job multiple times

If you want to carry out a parameter study using parameters generated externally, **zip** is the most appropriate factory type.

For **parameter sweep** and **zip** a set of jobs is created by substituting a range of values into a template job. Substitutions can be made in the command to be executed or the values obtained using environment variables.

When a workflow using job factory is submitted to PROMINENCE individual jobs will automatically be created. The job names will be of the form <workflow name>/<job name>/<id> where <id> is an integer.

A single workflow can contain multiple job factories applied to different jobs.

Parameter sweep

In this case numeric values are generated from start and end points in addition to an increment provided by the user.

Here is an example fragment which would need to be included in a workflow description:

```

"factories": [
  {
    "type": "parameterSweep",
    "name": "sweep",
    "jobs": [
      "<job-name>"
    ],
    "parameters": [
      {
        "name": "frame",
        "start": 1,
        "end": 4,
        "step": 1
      }
    ]
  }
]

```

(continues on next page)

(continued from previous page)

```
}
]
```

Here we specify the factory to be of type `parameterSweep`. The range of values used to create the jobs is defined in `parameters`. The name of the parameter is given by `name`. In this example the parameter `frame` is varied between the value `start` and at most `end` in increments of `step`. The factory is applied to all jobs specified in `jobs`.

Jobs can obtain the value of the parameter through the use of substitutions or environment variables. If a job's command was to include `$frame` or `${frame}`, this would be substituted by the appropriate value. An environment variable `PROMINENCE_PARAMETER_frame` would also be available to the job containing this value.

Additional parameters can be included in order to carry out multi-dimensional parameter sweeps.

If you wish to explicitly specify each value to be used, rather than specifying start and end values and a step, use a `zip` (described below) rather than a parametric sweep.

Zip

A set of jobs is created by substituting a range of values into a template job. The values to be used are specified in the form of lists. If multiple parameters are provided, the *i*-th job is provided with the *i*-th element from each list. The name comes from Python's `zip` function.

Here's an example fragment which would need to be included in a workflow description:

```
"factories": [
  {
    "name": "example",
    "jobs": [
      "<job-name>"
    ],
    "type": "zip",
    "parameters": [
      {
        "name": "start_value",
        "values": [
          0, 1, 2, 3
        ]
      },
      {
        "name": "end_value",
        "values": [
          8, 9, 10, 11
        ]
      }
    ]
  }
]
```

Here we specify the factory to be of type `zip`. The range of values used to create the jobs is defined in `parameters`. The name of each parameter is given by `name` and a list of values for each parameter is provided. In this example 4 jobs would be created, with:

- `start_value = 0`, `end_value = 8`
- `start_value = 1`, `end_value = 9`
- `start_value = 2`, `end_value = 10`

- start_value = 3, end_value = 11

Repeat

A set of identical jobs is created. The parameter `number` specifies the number of jobs to create from the template. Example:

```
"factories": [
  {
    "name": "example",
    "jobs": [
      "<job-name>"
    ],
    "type": "repeat",
    "number": 10
  }
]
```

Here 10 instances of the job with name `example` will be created.

5.1.4 Job factories with dependencies

Job factories can be applied to jobs in a directed acyclic graph.

In the example workflow description below we use a job factory to run 3 *process* jobs, then once these have completed a *merge* job is run:

```
{
  "name": "factory-dag-workflow",
  "jobs": [
    {
      "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
      },
      "tasks": [
        {
          "image": "busybox",
          "runtime": "singularity",
          "cmd": "echo $id"
        }
      ],
      "name": "process"
    },
    {
      "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
      },
      "tasks": [
        {
          "image": "busybox",
```

(continues on next page)

(continued from previous page)

```

        "runtime": "singularity",
        "cmd": "echo merge"
    }
],
    "name": "merge"
}
],
"factori  "factories": [
    {
        "name": "processing",
        "type": "parameterSweep",
        "jobs": [
            "process"
        ],
        "parameters": [
            {
                "name": "id",
                "start": 1,
                "end": 3,
                "step": 1
            }
        ]
    }
]
},
"dependencies": {
    "process": ["merge"]
}
}

```

5.2 Failures and retries

There are in general two classes of errors resulting in failing jobs: * Jobs which exited with an exit code of anything other than zero, * Jobs which failed due to infrastructure or network problems, or it was not possible to download input files or container images

By default the number of retries is zero, which means that if a job fails the workflow will fail. Any jobs which depend on a failed job will not be attempted. If the number of retries is set to one or more, if an individual job fails it will be retried up to the specified number of times. To set a maximum number of retries, include `maximumRetries` in the workflow definition, e.g.

```

"policies": {
    "maximumRetries": 2
}

```

The `rerun` command can be used to re-run any failed jobs in a workflow, for example:

```
prominence rerun <workflow id>
```

This will retry jobs which had previously failed, and execute any dependencies which were not run previously due to the failed jobs. This command can be used multiple times if necessary.

When the `rerun` command is used a new workflow id will be returned. After re-running a workflow the new workflow id should be used for checking the status. The original workflow id will only report the original number of successful jobs.

It is possible for jobs to access data from external storage systems in a POSIX-like way, like a standard filesystem. Users can specify the mount point to be used inside the container. [B2DROP](#), [OneData](#) (including EGI's [DataHub](#)) and WebDAV are supported.

Container images can be obtained from this storage if necessary. In this case the image should be in the form of a Singularity image or Docker archive.

6.1 B2DROP

[B2DROP](#), based on [Nextcloud](#), is a secure storage system accessible from any device and any location. 20 GB of storage is available to all researchers from the public B2DROP service and is integrated with EGI Check-in. Larger data volumes are available as a paid service and can be ordered through the [EOSC Marketplace](#).

The following JSON needs to be included in every job description where access to B2DROP is required:

```
"storage":{
  "type":"b2drop",
  "mountpoint":"/data",
  "b2drop":{
    "app-username":"***",
    "app-password":"***"
  }
}
```

where the app username and password should be set as appropriate. The mount point `/data` here is just an example and of course can be replaced with something else.

Note that the app username and password are not the same as the username and password used to access B2DROP. To create an app username and password, login to <https://b2drop.eudat.eu/>, then select **Settings** then **Security** and click **Create new app password**.

Using the PROMINENCE CLI to create jobs, the `--storage` option can be used to specify the name of a JSON file containing the above content. For example:

```
prominence create --storage my-b2drop.json ...
```

6.2 OneData

Storage accessible through OneData, including [EGI DataHub](#), can be made available to jobs. It should be noted that it is relatively straightforward to setup a OneData provider in order to integrate an existing storage system.

In order to mount your OneData storage in jobs firstly an access token needs to be created using the **Access tokens** menu in the OneData web interface.

The following JSON needs to be included in every job description where access to OneData is required:

```
"storage":{
  "type":"onedata",
  "mountpoint":"/data",
  "onedata":{
    "provider": "***",
    "token": "***"
  }
}
```

where the provider hostname and access token should be set as appropriate. The mount point `/data` here is just an example and can be replaced with something else.

6.3 WebDAV

The following JSON needs to be included in every job description where access to a storage system providing WebDAV is required:

```
"storage":{
  "type":"webdav",
  "mountpoint":"/data",
  "webdav":{
    "url": "***",
    "username": "***",
    "password": "***"
  }
}
```

In this case a URL is required in addition to a username and password.

CHAPTER 7

Using the API

PROMINENCE uses a RESTful API with data formatted in JSON. A POST request is used for job/workflow submission while GET requests are used to check the status of or retrieve information about jobs/workflows. A DELETE request is used to delete jobs/workflows.

An access token must be provided with each request in the `Authorization` header:

```
Authorization: Bearer <token>
```

where `<token>` should be replaced with the actual access token.

The base URL is <https://eosc.prominence.cloud/api/v1>, while the specific endpoints to use are as follows:

- For jobs: <https://eosc.prominence.cloud/api/v1/jobs>
- For workflows: <https://eosc.prominence.cloud/api/v1/workflows>

To begin with we will go through some basic examples using `cURL` in order to demonstrate common API requests and their responses, and then look at using the API with Python.

7.1 cURL

The `curl` command line tool can be used to submit jobs and check their status. Firstly, for simplicity we define an environment variable containing a valid token, e.g.

```
export ACCESS_TOKEN=<token>
```

where `<token>` should be replaced with the access token as mentioned previously.

7.1.1 Submitting a job

Create a file containing the JSON description of a job. In this example we use a file `testpi.json` containing the following:

```
{
  "resources": {
    "memory": 1,
    "cpus": 1,
    "nodes": 1,
    "disk": 10
  },
  "name": "calculate-pi",
  "tasks": [
    {
      "image": "eoscprominence/testpi",
      "runtime": "singularity"
    }
  ]
}
```

This job can be submitted by running the following command:

```
curl -i -X POST -H "Authorization: Bearer $ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d@testpi.json \
  https://eoscprominence.cloud/api/v1/jobs
```

If the submission was successful, this should return a response like the following:

```
HTTP/1.1 201 CREATED
Server: nginx/1.10.3 (Ubuntu)
Date: Fri, 05 Feb 2021 17:39:59 GMT
Content-Type: application/json
Content-Length: 12
Connection: keep-alive

{"id":1699}
```

We see here that the job id is 1699.

7.1.2 Checking the status of a job

We can check the status of this job with a simple GET request, here using `jq` to display the JSON in a pretty way:

```
curl -s -H "Authorization: Bearer $ACCESS_TOKEN" https://eoscprominence.cloud/api/v1/
↪ jobs/1169 | jq .
```

will return:

```
[
  {
    "events": {
      "createTime": 1612546799
    },
    "id": 1169,
    "name": "calculate-pi",
    "resources": {
      "cpus": 1,
      "disk": 10,
      "memory": 1,
```

(continues on next page)

(continued from previous page)

```

    "nodes": 1
  },
  "status": "pending",
  "tasks": [
    {
      "image": "eoscprominence/testpi",
      "runtime": "singularity"
    }
  ]
}
]

```

This request returns all information about the specified job. A completed job will have more information in the JSON response, for example:

```

curl -s -H "Authorization: Bearer $ACCESS_TOKEN" "https://eosc.prominence.cloud/api/v1/jobs/1181" | jq .

```

will return:

```

[
  {
    "events": {
      "createTime": 1612684304,
      "endTime": 1612684626,
      "startTime": 1612684595
    },
    "execution": {
      "maxMemoryUsageKB": 242814,
      "site": "OpenStack-UNIV-LILLE",
      "tasks": [
        {
          "cpuTimeUsage": 0.54800000000000012,
          "exitCode": 0,
          "imagePullStatus": "completed",
          "imagePullTime": 26.14915418624878,
          "maxResidentSetSizeKB": 63044,
          "retries": 0,
          "wallTimeUsage": 0.6737308502197266
        }
      ]
    },
    "id": 1181,
    "name": "calculate-pi",
    "resources": {
      "cpus": 1,
      "disk": 10,
      "memory": 1,
      "nodes": 1
    },
    "status": "completed",
    "tasks": [
      {
        "image": "eoscprominence/testpi",
        "runtime": "singularity"
      }
    ]
  }
]

```

(continues on next page)

(continued from previous page)

```
]
}
]
```

7.1.3 Listing all jobs

Alternatively we can list all currently active jobs, i.e. jobs which have not yet completed:

```
curl -s -H "Authorization: Bearer $ACCESS_TOKEN" https://eosc.prominence.cloud/api/v1/
↪ jobs | jq .
```

will return:

```
[
  {
    "events": {
      "createTime": 1612546799
    },
    "id": 1169,
    "name": "calculate-pi",
    "status": "pending",
    "tasks": [
      {
        "image": "eoscprominence/testpi",
        "runtime": "singularity"
      }
    ]
  }
]
```

7.1.4 Listing completed jobs

In order to list completed jobs (e.g. finished successfully, deleted, failed, or killed) add the query parameter completed with value true, for example:

```
curl -s -H "Authorization: Bearer $ACCESS_TOKEN" "https://eosc.prominence.cloud/api/
↪ v1/jobs?completed=true" | jq .
```

will return:

```
[
  {
    "events": {
      "createTime": 1612682844,
      "endTime": 1612683091
    },
    "id": 1179,
    "name": "calculate-pi",
    "status": "failed",
    "statusReason": "No matching resources currently available",
    "tasks": [
      {
        "image": "eoscprominence/testpi",
```

(continues on next page)

(continued from previous page)

```

        "runtime": "singularity"
    }
]
}
]

```

By default the last completed job will be shown. An additional query parameter `num` can be added specifying the number of jobs to display.

7.1.5 Getting the standard output or error from jobs

The following example returns the standard output from a job, in this case with id 1181:

```

curl -H "Authorization: Bearer $ACCESS_TOKEN" https://eosc.prominence.cloud/api/v1/
↪ jobs/1181/stdout

```

To get the standard error replace `stdout` above with `stderr`.

Note that the standard output and error can be obtained both once a job has completed and while it is running, so it is possible to watch what a job is doing in real time, no matter where in the world it is running.

7.1.6 Deleting jobs

Jobs can easily be deleted using the REST API, for example:

```

curl -i -H "Authorization: Bearer $ACCESS_TOKEN" -X DELETE https://eosc.prominence.
↪ cloud/api/v1/jobs/1169
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Date: Fri, 05 Feb 2021 18:01:28 GMT
Content-Type: application/json
Content-Length: 3
Connection: keep-alive

{}

```

7.2 Python

The standard `requests` module can be used to interact with the PROMINENCE service.

Below is a complete simple example which submits a basic job. A JSON description of the job is constructed and a HTTP POST request is used to submit the job to the PROMINENCE service. In order to authenticate with the PROMINENCE server the access token is read from a file (the same file used by the PROMINENCE CLI) and the appropriate header is constructed and included in the HTTP request.

```

import json
import os
import requests

# Define a job
job = {
    "resources": {

```

(continues on next page)

(continued from previous page)

```

        "memory": 1,
        "cpus": 1,
        "nodes": 1,
        "disk": 10
    },
    "name": "calculate-pi",
    "tasks": [
        {
            "image": "eoscprominence/testpi",
            "runtime": "singularity"
        }
    ]
}

# Read the access token
if os.path.isfile(os.path.expanduser('~/.prominence/token')):
    with open(os.path.expanduser('~/.prominence/token')) as json_data:
        data = json.load(json_data)

    if 'access_token' in data:
        token = data['access_token']
    else:
        print('The saved token file does not contain access_token')
        exit(1)

# Create the header including the auth token
headers = {'Authorization': 'Bearer %s' % token}

# Submit the job
response = requests.post('https://eoscprominence.cloud/api/v1/jobs', json=job,
↳ headers=headers)

# Check if the submission was successful and get the job id
if response.status_code == 201:
    if 'id' in response.json():
        print('Job submitted with id %d' % response.json()['id'])
else:
    print('Job submission failed with http status code %d and error: %s' % (response.
↳ status_code, response.text))

```

8.1 Jobs

8.1.1 LAMMPS on a single node

Here we run one of the [LAMMPS](#) benchmark problems using Intel's Singularity image. In this case we run an MPI job on a single node.

```
prominence create --cpus 4 \  
                  --memory 4 \  
                  --nodes 1 \  
                  --intelmpi \  
                  --artifact https://lammops.sandia.gov/inputs/in.lj.txt \  
                  --runtime singularity \  
                  shub://intel/HPC-containers-from-Intel:lammops \  
                  "/lammops/lmp_intel_cpu_intelmpi -in in.lj.txt"
```

This also illustrates using `--artifact` to download a file from a URL before executing the job. Note that it is not necessary to use *mpirun* to run the application as this is taken care of automatically.

8.2 Workflows

8.2.1 1D parameter sweep

Here we run a one-dimensional parameter sweep. Jobs are created where the variable `frame` is varied from 1 through to 4.

```
{  
  "name": "ps-workflow",  
  "jobs": [  
    {
```

(continues on next page)

(continued from previous page)

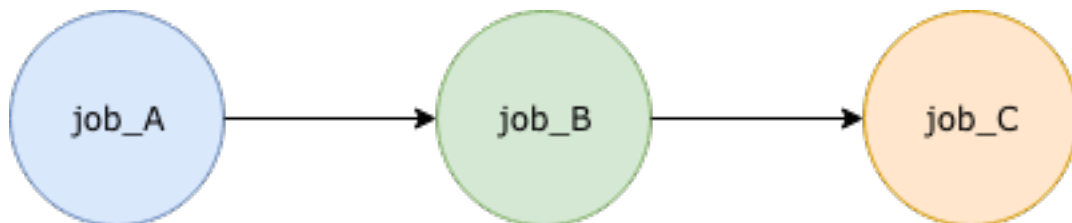
```

    "resources": {
      "nodes": 1,
      "cpus": 1,
      "memory": 1,
      "disk": 10
    },
    "tasks": [
      {
        "image": "busybox",
        "runtime": "singularity",
        "cmd": "echo $frame"
      }
    ],
    "name": "render"
  }
],
"factory": {
  "type": "parametricSweep",
  "parameters": [
    {
      "name": "frame",
      "start": 1,
      "end": 4,
      "step": 1
    }
  ]
}
]
}
}

```

8.2.2 Multiple steps

Here we consider a simple workflow consisting of multiple sequential steps, e.g.



In this example `job_A` will run first, followed by `job_B`, finally followed by `job_C`. A basic JSON description is shown below:

```

{
  "name": "multi-step-workflow",
  "jobs": [
    {
      "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
      },
      "tasks": [

```

(continues on next page)

(continued from previous page)

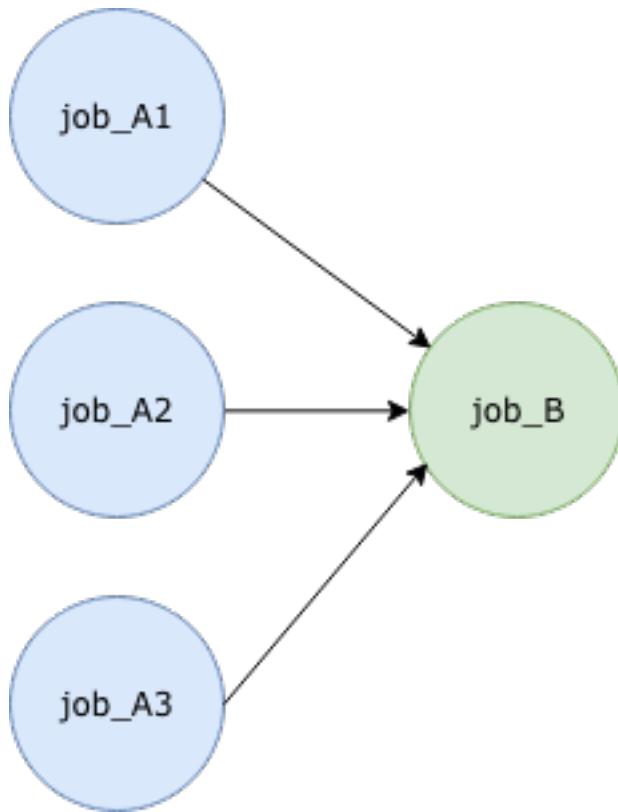
```

    {
      "image": "busybox",
      "runtime": "singularity",
      "cmd": "date"
    }
  ],
  "name": "job_A"
},
{
  "resources": {
    "nodes": 1,
    "cpus": 1,
    "memory": 1,
    "disk": 10
  },
  "tasks": [
    {
      "image": "busybox",
      "runtime": "singularity",
      "cmd": "date"
    }
  ],
  "name": "job_B"
},
{
  "resources": {
    "nodes": 1,
    "cpus": 1,
    "memory": 1,
    "disk": 10
  },
  "tasks": [
    {
      "image": "busybox",
      "runtime": "singularity",
      "cmd": "date"
    }
  ],
  "name": "job_C"
}
],
"dependencies": {
  "job_A": ["job_B"],
  "job_B": ["job_C"]
}
}

```

8.2.3 Scatter-gather

Here we consider the common type of workflow where a number of jobs can run in parallel. Once these jobs have completed another job will run. Typically this final step will take output generated from all the previous jobs. For example:



A basic JSON description is shown below:

```
{
  "name": "scatter-gather-workflow",
  "jobs": [
    {
      "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
      },
      "tasks": [
        {
          "image": "busybox",
          "runtime": "singularity",
          "cmd": "date"
        }
      ],
      "name": "job_A1"
    },
    {
      "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
      },
      "tasks": [
        {
```

(continues on next page)

(continued from previous page)

```

        "image": "busybox",
        "runtime": "singularity",
        "cmd": "date"
    }
],
    "name": "job_A2"
},
{
    "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
    },
    "tasks": [
        {
            "image": "busybox",
            "runtime": "singularity",
            "cmd": "date"
        }
    ],
    "name": "job_A3"
},
{
    "resources": {
        "nodes": 1,
        "cpus": 1,
        "memory": 1,
        "disk": 10
    },
    "tasks": [
        {
            "image": "busybox",
            "runtime": "singularity",
            "cmd": "date"
        }
    ],
    "name": "job_B"
}
],
"dependencies": {
    "job_A1": ["job_B"],
    "job_A2": ["job_B"],
    "job_A3": ["job_B"]
}
}

```

8.3 Jupyter notebooks

Since all interaction with PROMINENCE is via a REST API it is straightforward to use PROMINENCE from **any** Jupyter notebook. This can be done directly using the REST API, but here we make use of the PROMINENCE CLI.

Firstly install the PROMINENCE CLI:

```
!pip install prominence
```

Import the required module:

```
from prominence import ProminenceClient
```

Instantiate the PROMINENCE Client class, and obtain a token:

```
client = ProminenceClient()
client.authenticate_user()
```

As usual, you will be asked to visit a web page in your browser to authenticate. Note that the token retrieved is stored in memory and is not written to disk. If the token expires you will need to re-run `authenticate_user()`.

Construct the JSON job description. In this example we use OSPRay to render an image:

```
# Required resources
resources = {
    'cpus': 16,
    'memory': 16,
    'disk': 10,
    'nodes': 1
}

# Define a task
task = {
    'image': 'alahiff/ospray',
    'runtime': 'singularity',
    'cmd': '/opt/ospray-1.7.1.x86_64.linux/bin/ospBenchmark --file NASA-B-field-sun.
↪osx --renderer scivis -hd --filmic -sg:spp=8 -i NASA'
}

# Output files
output_files = ['NASA.ppm']

# Input files (artifacts)
artifact = {'url': 'http://www.sdvis.org/ospray/download/demos/NASA-B-field-sun/NASA-B-
↪field-sun.osx'}

# Create a job
job = {
    'name': 'NASAstreamlines',
    'resources': resources,
    'outputFiles': output_files,
    'artifacts': [artifact],
    'tasks': [task]
}
```

Now submit the job:

```
id = client.create_job(job)
print('Job submitted with id', id)
```

Jobs can be listed and the status of jobs checked. Here are some examples:

```
# Check status of a particular job
job = client.describe_job(387)
```

(continues on next page)

(continued from previous page)

```
print('Job status is', job['status'])

# List currently active jobs
print(client.list_jobs())

# List last completed job
print(client.list_jobs(completed=True))

# List the last 4 completed jobs
print(client.list_jobs(completed=True, num=4))

# List all jobs with label app=hello
print(client.list_jobs(all=True, constraint='app=hello'))
```

PROMINENCE was originally developed in the Fusion Science Demonstrator in EOSCpilot. The European Open Science Cloud for Research pilot project was funded by the European Commission, DG Research & Innovation under contract no. 739563. This work is also co-funded by the EOSC-hub project (Horizon 2020) under Grant number 777536.

